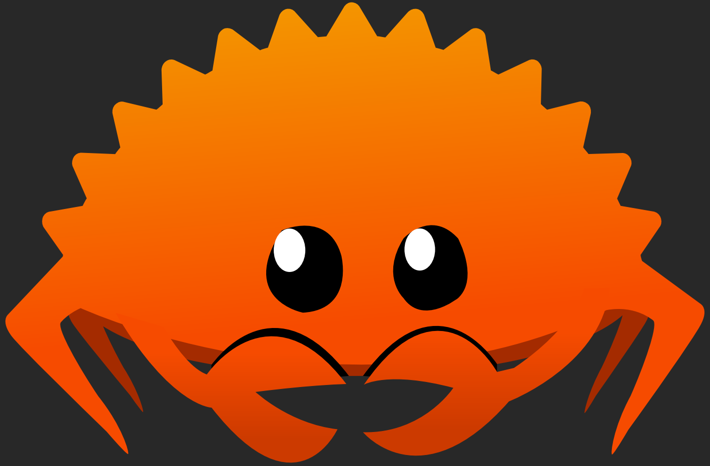


Rust

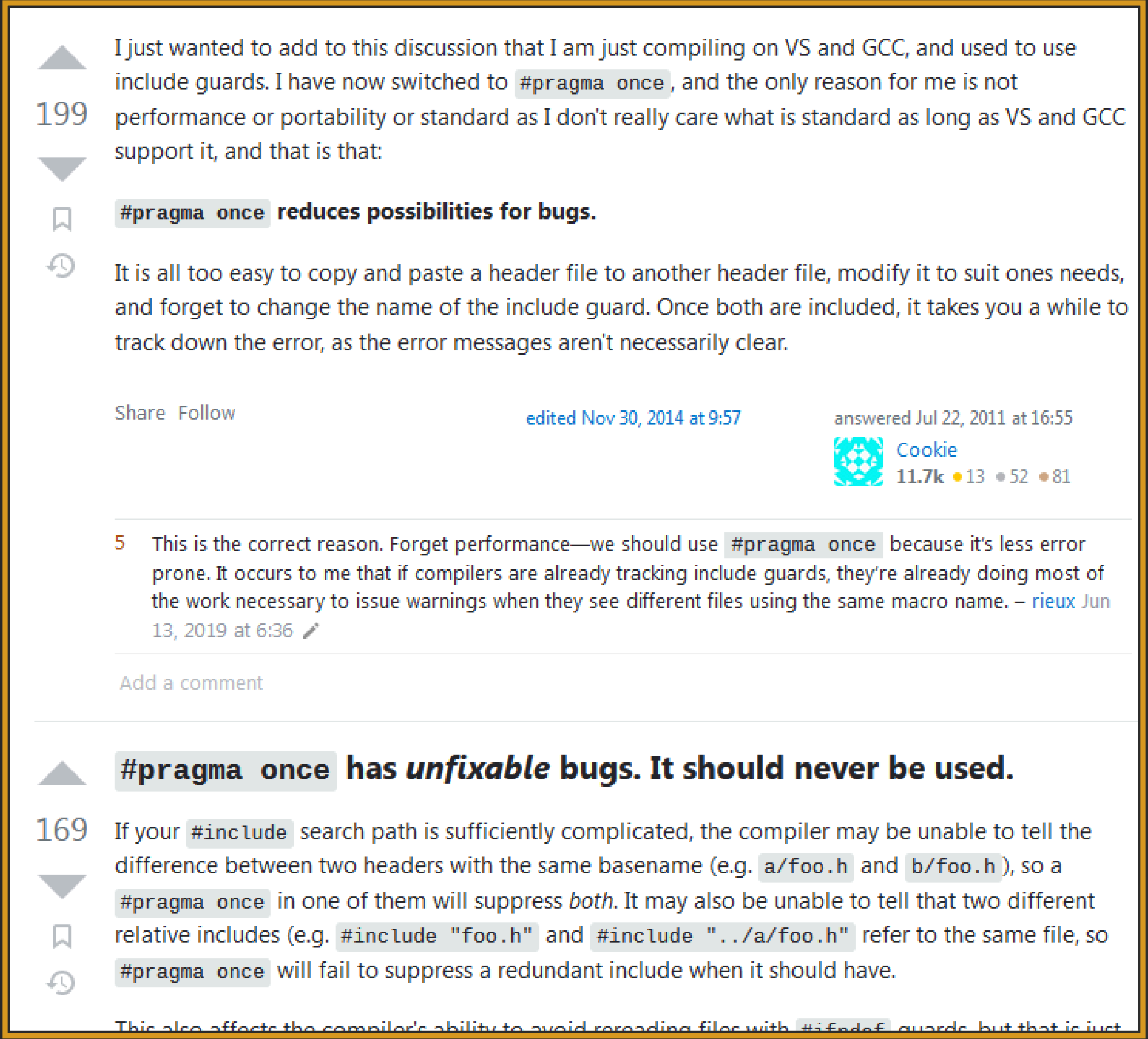
O zarządzaniu projektem, łączeniu z innymi językami i o tym kiedy przepisywać stary kod

Łukasz Neumann

lukasz.neumann@pw.edu.pl

https://staff.elka.pw.edu.pl/~lneumann/rust_3.pdf

Po co są pliki nagłówkowe (`.h`)?



199

I just wanted to add to this discussion that I am just compiling on VS and GCC, and used to use include guards. I have now switched to `#pragma once`, and the only reason for me is not performance or portability or standard as I don't really care what is standard as long as VS and GCC support it, and that is that:

`#pragma once` reduces possibilities for bugs.

It is all too easy to copy and paste a header file to another header file, modify it to suit ones needs, and forget to change the name of the include guard. Once both are included, it takes you a while to track down the error, as the error messages aren't necessarily clear.

Share Follow

edited Nov 30, 2014 at 9:57

answered Jul 22, 2011 at 16:55
Cookie
11.7k ●13 ●52 ●81

5 This is the correct reason. Forget performance—we should use `#pragma once` because it's less error prone. It occurs to me that if compilers are already tracking include guards, they're already doing most of the work necessary to issue warnings when they see different files using the same macro name. – rieux Jun 13, 2019 at 6:36

Add a comment

169

`#pragma once` has *unfixable* bugs. It should never be used.

If your `#include` search path is sufficiently complicated, the compiler may be unable to tell the difference between two headers with the same basename (e.g. `a/foo.h` and `b/foo.h`), so a `#pragma once` in one of them will suppress *both*. It may also be unable to tell that two different relative includes (e.g. `#include "foo.h"` and `#include "../a/foo.h"` refer to the same file, so `#pragma once` will fail to suppress a redundant include when it should have.

This also affects the compiler's ability to avoid rereading files with `#ifndef` guards, but that is just

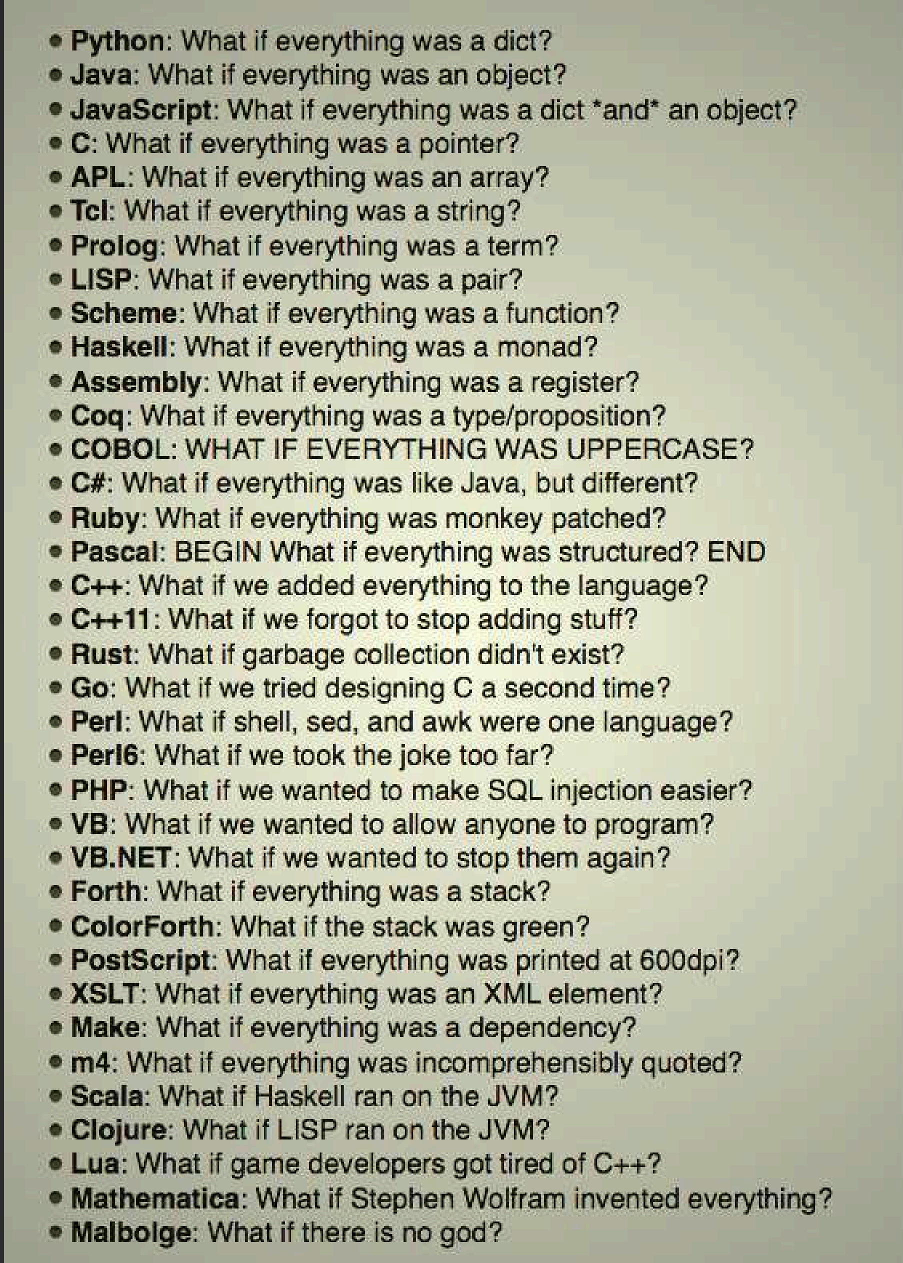
Moduły w Ruście

- brak plików nagłówkowych
- cztery poziomy abstrakcji
    - `path` - ścieżka, nazwana rzecz, np. moduł, plik, funkcja itd.
    - `module` - moduł, pozwala na kontrolowanie zakresu widoczności ścieżek
        - może być plikiem, katalogiem lub kawałkiem pliku (jako `mod foo { ... }`)
    - `crate` - zbiór modułów, najmniejsza jednostka rozpatrywana przez kompilator
        - `binary crate` - program wykonywalny
        - `library crate` - biblioteka
    - `package` - pakiet, zbiór “skrzyń”

Cargo live demo

https://gitlab-stud.elka.pw.edu.pl/lneumann/cargo-fractals-demo

Po co łączyć języki programowania?


- **Python**: What if everything was a dict?
- **Java**: What if everything was an object?
- **JavaScript**: What if everything was a dict *and* an object?
- **C**: What if everything was a pointer?
- **APL**: What if everything was an array?
- **Tcl**: What if everything was a string?
- **Prolog**: What if everything was a term?
- **LISP**: What if everything was a pair?
- **Scheme**: What if everything was a function?
- **Haskell**: What if everything was a monad?
- **Assembly**: What if everything was a register?
- **Coq**: What if everything was a type/proposition?
- **COBOL**: WHAT IF EVERYTHING WAS UPPERCASE?
- **C#**: What if everything was like Java, but different?
- **Ruby**: What if everything was monkey patched?
- **Pascal**: BEGIN What if everything was structured? END
- **C++**: What if we added everything to the language?
- **C++11**: What if we forgot to stop adding stuff?
- **Rust**: What if garbage collection didn't exist?
- **Go**: What if we tried designing C a second time?
- **Perl**: What if shell, sed, and awk were one language?
- **Perl6**: What if we took the joke too far?
- **PHP**: What if we wanted to make SQL injection easier?
- **VB**: What if we wanted to allow anyone to program?
- **VB.NET**: What if we wanted to stop them again?
- **Forth**: What if everything was a stack?
- **ColorForth**: What if the stack was green?
- **PostScript**: What if everything was printed at 600dpi?
- **XSLT**: What if everything was an XML element?
- **Make**: What if everything was a dependency?
- **m4**: What if everything was incomprehensibly quoted?
- **Scala**: What if Haskell ran on the JVM?
- **Clojure**: What if LISP ran on the JVM?
- **Lua**: What if game developers got tired of C++?
- **Mathematica**: What if Stephen Wolfram invented everything?
- **Malbolge**: What if there is no god?

Jak łączyć języki programowania?

- Inter-process Communication (IPC)
    - pliki
    - sygnały
    - gniazda
    - kolejki
    - pamięć współdzielona
- Virtual Machine (VM)
- Foreign Function Interface (FFI)

Foreign Function Interface (FFI)

- mechanizm, dzięki któremu kod napisany w języku `X` może wywołać kod napisany w języku `Y`
- żeby zapewnić FFI należy wziąć pod uwagę następujące kwestie:
  - współpraca ze środowiskiem języka `Y` (np. z interpreterem) i/lub zgodność z Application Binary Interface (ABI)
  - poprawne zarządzanie wspólnymi zasobami
    - współdzielenie i mutowalność
    - zwalnianie
  - mapowanie typów między językami
- C jest obecnie wzorcem na podstawie którego buduje się FFI

FFI w praktyce

```rust
use libc {size_t, c_int, c_double};

/// Wrapper around C FFI for the
/// evaluate function
#[no_mangle]
pub extern "C" fn evaluate(
    individual_ptr: *const c_int,
    individual_length: size_t,
) -> c_double {
    // ...
}
```

```python
from ctypes import (
    POINTER, c_double, c_int, c_size_t, cdll
)

lib = cdll.LoadLibrary(rust_lib_path)

lib.evaluate.argtypes = [POINTER(c_int), c_size_t]
lib.evaluate.restype = c_double

def evaluate(array):
    c_array = (c_int * len(array))(*array)
    return lib.evaluate(c_array, len(array))
```

C++ i Pybind11

```cpp
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring

    m.def("add", &add, "A function that adds two numbers");
}
```

```python
from example import add

assert add(1, 2) == 3
```

Rust + PyO3

```rust
use pyo3::prelude::*;

/// Formats the sum of two numbers as string.
#[pyfunction]
fn squared_from_string(a: &str) -> PyResult<f32> {
    Ok(a.parse::<f32>()?.powf(2.))
}

/// A Python module implemented in Rust.
#[pymodule]
fn squarifier(m: &Bound<'_, PyModule>) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(squared_from_string, m)?)?;
    Ok(())
}
```

```python
from squarifier import squared_from_string

assert squared_from_string("7") == 49.0
```

PyO3 live demo

https://gitlab-stud.elka.pw.edu.pl/lneumann/pyo3-with-benchmarking-demo

Kiedy przepisywać projekt w innym języku?

- Obecny system **nie spełnia wymagań** (np. za duże zużycie zasobów bądź zbyt wolne działanie)
- Obecny system nie pozwala na sensowną rozbudowę o nową funkcjonalność ze względu na **architekturę**
- Język w jakim napisany jest obecny system sprzyja **notorycznym bugom**, w szczególności związanych z bezpieczeństwem i stabilnością
- Język/technologia o którą oparty jest nasz projekt **przestaje być wspierana**
- Nie ma **alternatywnych narzędzi**, które rozwiązują nasz problem

Jak ryzykowne jest przepisywanie kodu?

> Netscape 6.0 is finally going into its first public beta. There never was a version 5.0. The last major release, version 4.0, was released almost three years ago. Three years is an awfully long time in the Internet world. During this time, Netscape sat by, helplessly, as their market share plummeted.
>
> It’s a bit smarmy of me to criticize them for waiting so long between releases. They didn’t do it on purpose, now, did they?
>
> Well, yes. They did. They did it by making the **single worst strategic mistake** that any software company can make:
>
> They decided to rewrite the code from scratch.

https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/

Na co zwracać uwagę przy wyborze nowego języka programowania?

- “Komfort” programisty
  - Znajomość języka przez zespół
  - Dostępność narzędzi i bibliotek w danym języku
  - Szybkość pisania kodu
  - Utrzymywalność kodu
- Wymagania sprzętowe/systemowe
  - Dostępność języka na danej platformie
  - Wydajność języka (m.in. czas działania i zużycie pamięci)
- Stabilność języka

Rewrite checklist by Dropbox

- **Have you exhausted incremental improvements?**
  - Have you tried **refactoring code** into better modules?
  - Have you tried improving performance by **optimizing hotspots**?
  - Can you deliver **incremental value**?
- **Can you pull off a rewrite?**
  - Do you deeply **understand** and respect the **current system**?
  - Do you have the **engineering-hours**?
  - Can you accept a **slower** rate of **feature development**?
- **Do you know what you're going towards?**
  - Why will it be better the second time?
  - What are your **principles** for the new system?

https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine

Case study: Akita Agent

Last summer, my team and I faced a question many young startups face: Should we rewrite our system in Rust?

At the time of the decision, we were primarily writing in Go. I was working on an agent that passively watches network traffic, parses API calls and sends obfuscated summaries back to our service for analysis. **As users were starting to run more traffic through us, memory usage by the agent grew to an unacceptably high level, impacting performance**.

This led me to spend 25 days in despair and immerse myself in the details of Go's memory management, our technology stack and the profiling tools available — **trying to get our memory footprint back under control**. Go's fully automatic memory management makes this no easy feat.

Spoiler: I emerged victorious and **our team still uses Go**.

Case study: Cloudflare and Pingora

Over the past few years, as we've continued to grow our customer base and feature set, **we continually evaluated three choices**:

1. *Continue to invest in NGINX* and possibly fork it to tailor it 100% to our needs. We had the expertise needed, but given the architecture limitations mentioned above, significant effort would be required to rebuild it in a way that fully supported our needs.

2. *Migrate to another 3rd party proxy codebase*. [...] But this path means the same cycle may repeat in a few years.

3. *Start with a clean slate*, building an in-house platform and framework. This choice requires the most upfront investment in terms of engineering effort.

We evaluated each of these options every quarter for the past few years. **There is no obvious formula to tell which choice is the best.** For several years, we continued with the path of the least resistance, continuing to augment NGINX. However, at some point, building our own proxy's return on investment seemed worth it. We made a call to build a proxy from scratch [...]

Jak przepisywać projekt?

- Inkrementalnie
- ALL IN
- W obu przypadkach należy przygotować:
    - Ekstensywne testy
    - Benchmarki
    - Telemetrię/logowanie (zwłaszcza w przypadku optymalizacji zasobów)

Przepisywanie inkrementalne

- Podejście używane przy w miarę **sensownej architekturze projektu**
- Przepisujemy wybrane moduły, najlepiej z jak *najmniejszą liczbą zależności*
- Przepisany moduł **testujemy** i **benchmarkujemy**, zamieniamy ze starym modułem i *obserwujemy działanie*
- Nowe moduły powstają w nowym języku
- Zalety
  - Szybko można zewaluować poprawę działania projektu
  - Czasem pozwala na szybką poprawę czasu działania/zużycia zasobów jeśli zaczynamy od przepisania wąskich gardeł
  - Pozwala na inkrementalną naukę nowego języka przez kolejnych członków zespołu
- Wady
  - Łączenie dwóch języków wymaga większego nakładu pracy i potencjalnie wprowadza regresje w czasie działania

Case study: Discord and Read State service

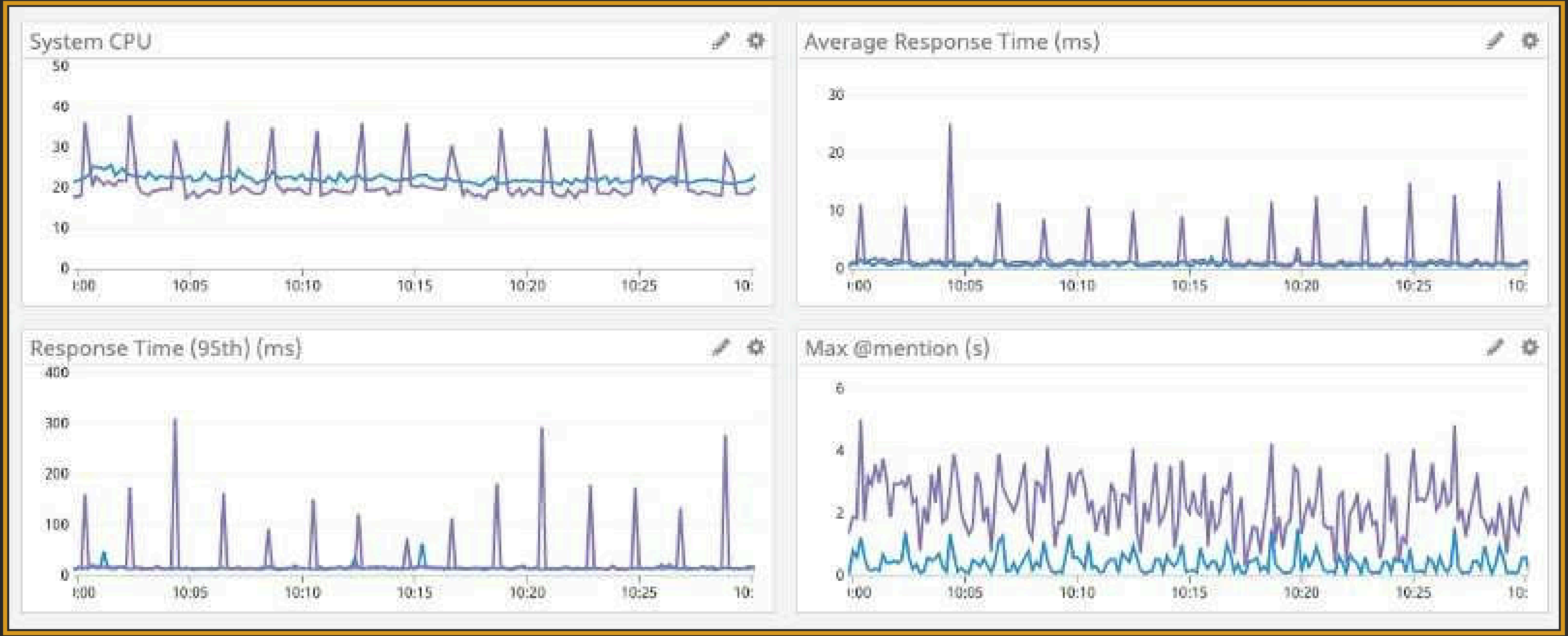
After digging through the Go source code, we learned that **Go will force a garbage collection run every 2 minutes at minimum**. In other words, if garbage collection has not run for 2 minutes, regardless of heap growth, go will still force a garbage collection.

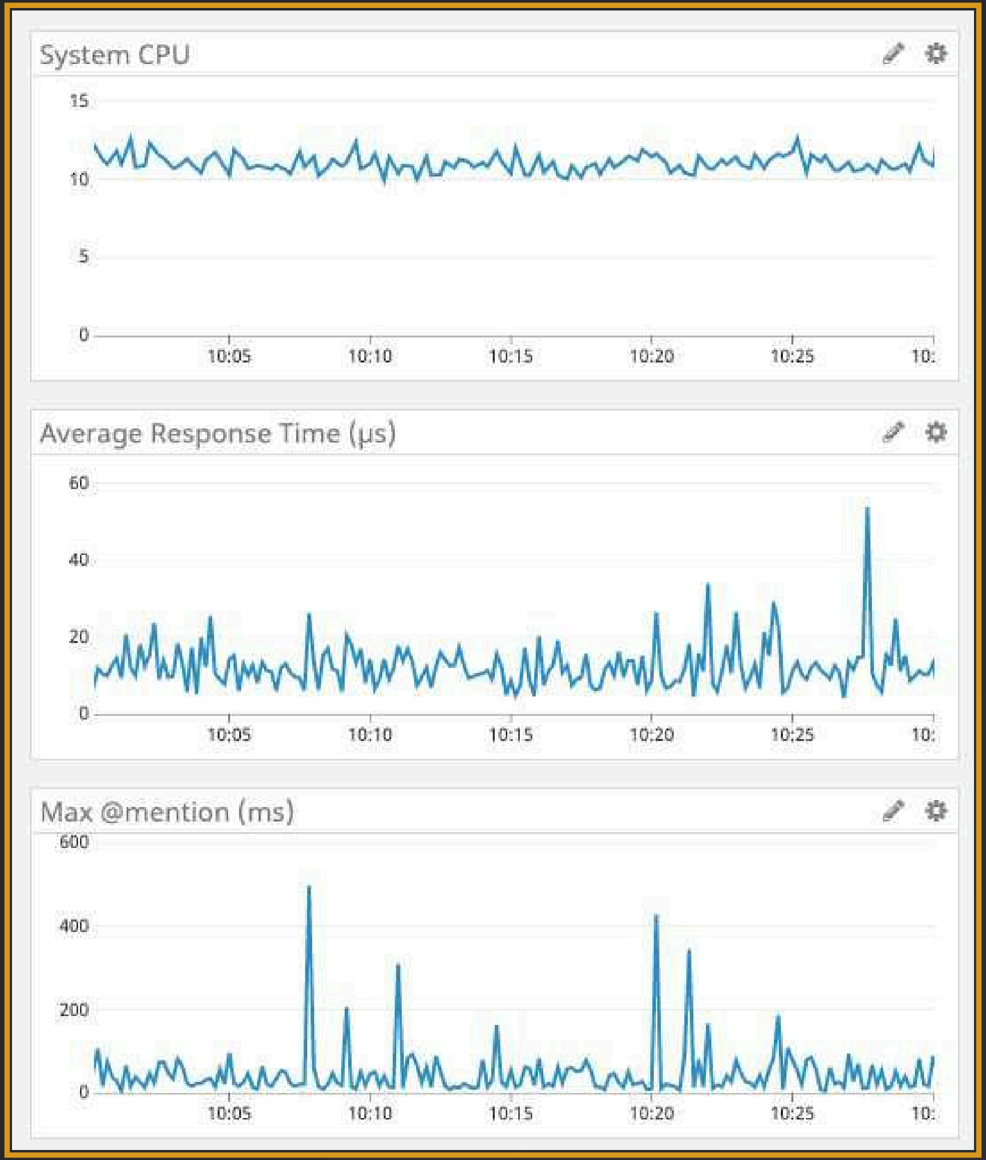
We figured we could **tune the garbage collector** to happen more often in order to prevent large spikes [...]

After a significant amount of load testing different cache capacities, we **found a setting that seemed okay**. Not completely satisfied, but satisfied enough and with bigger fish to fry, we left the service running like this for **quite some time**.

During that time we were seeing more and more success with Rust in other parts of Discord and we collectively decided we wanted to create the frameworks and libraries needed to build new services fully in Rust. This service was a great candidate to port to Rust since it was **small** and **self-contained**, but we also hoped that Rust would **fix these latency spikes**. So we took on the task of porting Read States to Rust

https://discord.com/blog/why-discord-is-switching-from-go-to-rust


System CPU
50
40
30
20
10
0
:00
10:05
10:10
10:15
10:20
10:25
10:
Average Response Time (ms)
30
20
10
0
:00
10:05
10:10
10:15
10:20
10:25
10:
Response Time (95th) (ms)
400
300
200
100
0
:00
10:05
10:10
10:15
10:20
10:25
10:
Max @mention (s)
6
4
2
0
:00
10:05
10:10
10:15
10:20
10:25
10:


System CPU
15
10
5
0
10:05
10:10
10:15
10:20
10:25
10:
Average Response Time (µs)
60
40
20
0
10:05
10:10
10:15
10:20
10:25
10:
Max @mention (ms)
600
400
200
0
10:05
10:10
10:15
10:20
10:25
10:

Case study: Fish shell

We've experienced some pain with C++. In short:

- tools and compiler/platform differences
- ergonomics and (thread) safety
- community

Frankly, the **tooling around the language isn't good** […] This means adopting recent C++ standards would complicate the lives of packagers and would-be contributors. For example, we started using C++11 in 2016, and yet we still needed to upgrade the compilers on our build machines until 2020. […]

Fish also uses threads for […] autosuggestions and syntax highlighting, and one long-term project is to **add concurrency to the language**. […]

We prototyped true multithreaded execution in C++, but it just didn't work out. For example, it was too easy to accidentally share objects across threads, with only post-hoc tools like Thread Sanitizer to prevent it.

The ergonomics of C++ are also simply not good - header files are annoying, templates are complicated, you can easily cause a compile error that throws pages of overloads in the standard library at you. Many functions are unsafe to use. […]

https://fishshell.com/blog/rustport/

Case study: Fish shell

And the **standard prioritizes performance over ergonomics**. Consider for instance string_view, which provides a non-owning slice of a string. This is an extremely modern, well-liked feature that C++ programmers often claim is a great reason to switch to C++17. And it is **extremely easy to run into use-after-free bugs** with it, because the ergonomics weren't a priority.

Finally, subjectively, **C++ isn't drawing in the crowds**. We have never had a lot of C++ contributors. Over the 11 years fish used C++, only 17 people have at least 10 commits to the C++ code. We also don't know a lot of people who would love to work on a C++ codebase in their free time.

We need to get one thing out of the way: **Rust is cool. It's fun.**

It's tempting to try to sweep this under the rug because it feels gauche to say, but it's actually important for a number of reasons.

For one, fish is a hobby project, and that means we want it to be fun for us. **Nobody is being paid to work on fish, so we need it to be fun. Being fun and interesting also attracts contributors.**

Rust also has **great tooling**. [...] And it is *easy* to get that tooling installed

https://fishshell.com/blog/rustport/

Case study: Fish shell

Rust has **great ergonomics** - the difference between C++'s pointers (which can always be NULL) and Rust's Options are apparent very quickly even to those of us who had never used it before. We did have a backport of C++'s optional, and liked using it, but it was never as integrated as Rust's Options were.

Having an explicit use system where you know exactly which function comes from which module is a great improvement over `#include`.

Rust makes it **nice to add dependencies**. [...]

But the **killer feature** of Rust, *from fish-shell's perspective*, **is Send and Sync**, statically enforcing rules around threading. "Fearless concurrency" is too strong - you can still blow your leg off with fork or signal handlers - but Send and Sync will be the key to unlocking fully multithreaded execution, with confidence in its correctness.

We *did not do a comprehensive survey of other languages*. We were confident Rust was up to the task and either already knew it or wanted to learn it, so we picked it.

https://fishshell.com/blog/rustport/

Case study: Fish shell

We had decided we were gonna do a *“Fish Of Theseus”* port - we would move over, component by component, until no C++ was left. And **at every stage of that process, it would remain a working fish**.

This was a necessity - if we didn’t, we would not have a working program for months, which is not only demoralizing but would also have precluded us from using most of *our test suite* - which is *end-to-end tests that run a script or fake a terminal interaction*. We would also not have been able to do another C++ release, putting some cool improvements into the hands of our users.

Had we chosen to *disappear into a hole we might not have finished at all*, and we would have to re-do a bunch of work once it became testable. We also **mostly kept the structure of the C++ code intact** - if a function is in the “env” subsystem, it would stay there. Resisting the temptation to clean up allowed us to compare the before and after to find places where we had mistranslated something.

https://fishshell.com/blog/rustport/

Case study: Fish shell

We've succeeded. This was a gigantic project and we made it. The sheer scale of this is perhaps best expressed in numbers:

- 1155 files changed, 110247 insertions(+), 88941 deletions(-) (excluding translations)
- 2604 commits by over 200 authors
- 498 issues
- Almost 2 years of work
- 57K Lines of C++ to 75K Lines of Rust (plus 400 lines of C)

The **beta works very well**. Performance is usually slightly better in terms of time taken, memory use has a slightly higher floor but a lower ceiling [...] These things can all be improved, of course, but **for a first result it is encouraging**.

Fish is still a bit of an odd duck...fish as a Rust program. It has **some bits that smell like C** spirit, directly using the C API [...]. It still uses UTF-32 strings [...]. We hope to find a nicer solution here, but it wasn't necessary for the first release.

The port wasn't without challenges, and it did not all go entirely as planned. But overall, it went pretty dang well. We're now left with a **codebase that we like a lot more**, that has already **gained some features that would have been much more annoying to add with C++**, with more on the way, and we did it while creating a separate 3.7 release that also included some cool stuff.

And **we had fun doing it**.

https://fishshell.com/blog/rustport/

All in - przepisanie projektu od zera

- Pozwala na gruntowną zmianę architektury projektu, często zgodną z filozofią programowania w nowym języku
- Podczas przepisywania można wprowadzić tzw. feature freeze w starej wersji projektu
- Wymaga bardzo dobrych testów integracyjnych
- Cały zespół musi znać nowy język programowania

Case study: Arti

Why rewrite Tor in Rust? Because despite (or because of) its maturity, the C Tor implementation is showing its age. While C was a reasonable choice back when we started working on Tor 2001, we've always suffered from its limitations: **it encourages a needlessly low-level approach to many programming problems, and using it safely requires painstaking care and effort**. Because of these limitations, that pace of development in C has always been slower than we would have liked.

What's more, our **existing C implementation has grown over the years to have a not-so-modular design**: nearly everything is connected to everything else, which makes it even more difficult to analyze the code and make safe improvements.

In 2017, we started experimenting with adding Rust inside the C Tor codebase, with a view to replacing the code bit by bit. One thing that we found, however, was that our existing C code was not modular enough to be easily rewritten.[...] The parts of the code that were isolated enough to replace were mostly trivial, and seemed not worth the effort—whereas the parts that most needed replacement were to intertwined with each other to practically disentangle. **We tried to disentangle our modules, but it proved impractical to do so without destabilizing the codebase**.

https://blog.torproject.org/arti_100_released/

Case study: Arti

- At every stage, we've encountered **way fewer bugs** than during comparable C development. The bugs that we have encountered have almost all been semantic/algorithmic mistakes (real programming issues), not mistakes in using the Rust language and its facilities. [...]
- **Development** of comparable features has gone **way faster**, even considering that we're building most things for the second time. Some of the speed improvement is due to Rust's more expressive semantics and more usable library ecosystem—but a great deal is due to the confidence Rust's safety brings.
- Portability has been far easier than C, though sometimes we're forced to deal with differences between operating systems. [...]
- One still-uncracked challenge is *binary size*. [...]
- We've found that Arti has attracted **volunteer contributions in greater volume and with less friction than C Tor**. New contributors are greatly assisted by Rust's strong type system, excellent API documentation support, and safety properties. These features help them find where to make a change, and also enable making changes to unfamiliar code with much greater confidence.

https://blog.torproject.org/arti_100_released/

Case study: Architect of Ruin

When I started building Architect of Ruin in *December 2023* I chose to build it in the Bevy game engine. My choice was motivated by a *personal interest in Rust* – a language I derive a lot of joy in using. This was furthered by Bevy's ECS model which I also find *fun to work with* and the openness of *Bevy's community* which I have a genuine appreciation for.

I want to begin by stating that **I anticipated many of these challenges before they manifested**. [...] My love of Rust and Bevy meant that I would be willing to bear some pain that other game developers might choose to avoid. I didn't walk blindly into these specific problems, but they bit harder than I was expecting.

Collaboration - I started this project with my brother. [...] he's new to coding. Onboarding him directly into game dev while simultaneously navigating Rust's unique aspects proved challenging.

https://deadmoney.gg/news/articles/migrating-away-from-rust

Case study: Architect of Ruin

Abstraction - While my initial motivation was the enjoyment of Rust, the project's bottleneck increasingly became the rapid iteration of higher-level gameplay mechanics. As the codebase grew, we found that translating gameplay ideas into code was less direct than we hoped. Rust's (powerful) low-level focus didn't always lend itself to a flexible high-level scripting style needed for rapid prototyping within our specific gameplay architecture. [...]

Migration - Bevy is young and changes quickly. Each update brought with it incredible features, but also a substantial amount of API thrash. As the project grew in size, the burden of update migration also grew. Minor regressions were common in core Bevy systems (such as sprite rendering), and these led to moments of significant friction and unexpected debugging effort.

https://deadmoney.gg/news/articles/migrating-away-from-rust

Case study: Architect of Ruin

Learning - [...] I regularly use AI to learn new technologies, discuss methods and techniques, review code, etc. The maturity and vast amount of stable historical data for C# and the Unity API mean that tools like Gemini consistently provide highly relevant guidance. While Bevy and Rust evolve rapidly - which is exciting and motivating - the pace means AI knowledge lags behind, reducing the efficiency gains I have come to expect from AI assisted development. [...]

Modding - [...] I came to understand many inherent limitations in Rust and Bevy that would make the task more difficult. Lack of a clear solution to scripting and an unstable ABI (application binary interface) raised concerns. [...]

These factors combined - **the desire for a smoother workflow across experience levels, the need for a high-level abstraction for gameplay, optimizing productivity, and modding** - pointed towards a re-evaluation of the project's next phase.

https://deadmoney.gg/news/articles/migrating-away-from-rust

Case study: Architect of Ruin

In the first week of *January of 2025*, Blake and I decided to do a **cost-benefit analysis**. We wrote down all the options: Unreal, Unity, Godot, continuing in Bevy, or rolling our own. **We wrote extensive pros and cons**, emphasizing how each option fared by the criteria above: Collaboration, Abstraction, Migration, Learning, and Modding.

The team decided to invest in an experiment. I would pick three core features and see how difficult they would be to implement in Unity. We would spend no more than 3 weeks on the task. **We would invest 10% of effort to see if we should invest the other 90% in a full port**.

We finished all three tasks in 3 days!

https://deadmoney.gg/news/articles/migrating-away-from-rust

Case study: Architect of Ruin

Code size shrank substantially, massively improving maintainability. As far as I can tell, most of this savings was just in the elimination of ECS boilerplate.

Everything felt tighter and more straightforward. **Update migration anxiety was gone** [...]

The shift has measurably **improved our day-to-day development**. **Iteration feels faster**, allowing ideas to flow into the game more easily. We've also been able to **leverage ecosystem tools** [...]

Sometimes you have to burn time to earn time. I think **we are way ahead** of where we would have been had we stuck with Bevy. Our agility in implementing rendering features while also pushing gameplay forward is much higher.

https://deadmoney.gg/news/articles/migrating-away-from-rust

Case study: Asciinema

Long story short: asciinema-player has been reimplemented from scratch in JavaScript and Rust, resulting in *50x faster virtual terminal interpreter*, while at the same time, *reducing the size of the JS bundle 4x*.

You may wonder what prompted the move from the previous ClojureScript implementation. As much as I love Clojure/ClojureScript there were several major and minor problems I couldn't solve, mostly around these 3 areas: speed, size, integration with JS ecosystem [...]

It [the player] exercised many types of optimizations, like memoization (trading memory for CPU time) and run-ahead (which used a lot of memory by precomputing terminal contents for each future frame).

At first I planned to implement the terminal emulation part in Rust without any optimizations, just write idiomatic Rust code, then revisit the tricks from the old implementation. The initial benchmarks blew my mind though, showing that **spending additional time on optimizing the emulation part is absolutely unnecessary**.

https://blog.asciinema.org/post/smaller-faster/

Dodatkowe referencje

- https://www.maturin.rs/
- https://dropbox.tech/application/why-we-built-a-custom-rust-library-for-capture
- https://ohadravid.github.io/posts/2023-03-rusty-python/
- https://github.com/tensorflow/tensorboard/issues/4784
- https://blog.cloudflare.com/building-fast-interpreters-in-rust/
- https://reddit.com/r/rust/comments/1457x29/what_are_the_scenarios_where_rewrite_it_in_rust/